广东工业大学

计算机学院

《图形学与虚拟现实》

实验报告

课 程 名 称：	图形学与虚拟现实
实 验 名 称：	使用 OpenGL 实现天空盒中物体建模渲染、环境贴图与视角变换
指 导 教 师：	战荫伟
学 生 姓 名：	欧炜标
班 级 学 号：	3121005358
实 验 日 期：	2024/5/28
实 验 成 绩：	

目录

1 实验概述

实验的主要内容是在 VS2022 中通过使用 OpenGL 工具，对一基本几何体进行建模，并进行渲染等过程，包括一下六个部分：

1. 实验概述
2. OpenGL 环境配置：使用 vcpkg（Visual C++ Package Manager）在 VS2022 中配置 OpenGL 开发环境，安装了 GLEW、GLFW、GLM、SOIL2 等必要的库，并集成至 Visual Studio。
3. 构建几何模型——环形：基于 Paul Baker 的策略，实现了一个名为 Torus 的类，用于创建和绘制圆环。该类计算顶点坐标、纹理坐标、法线以及切向量，并使用这些数据生成环面的三角形网格。
4. 增加光照处理：为增强场景的深度感和立体感，实验中增加了光照处理，实现了环境光反射（Ambient）、漫反射（Diffuse）和镜面反射（Specular）的 ADS 模型，并使用 Gouraud 着色技术进行平滑渲染。
5. 天空盒实现与模型贴图：
   1) 天空盒实现：创建了一个立方体对象，并使用 OpenGL 的纹理立方体贴图法，通过 SOIL2 库读取六个面的纹理图像，构建了一个能够围绕相机放置的全景天空盒。
   2) 环境贴图实现：采用视图向量和法向量组合计算反射向量，并使用该反射向量从立方体贴图中查找元素，实现了环境贴图，增加了物体表面的反射效果。
6. 交互设置：为了实现全景特性，实验中增加了鼠标回调函数和多摄像机设置，允许用户通过鼠标操作和右键切换来实现视角变换，观察 360° 全景视图。
7. 实验总结

2 OpenGL 环境配置

VS2022 中搭建 OpenGL 环境，包括 GLEW 基本库、GLFW 窗口管理库、GLM 数学运算库和 SOIL2 图像加载库。

按照常规的方法，在 VS 中配置 OpenGL 环境需要从网上下载各个库的源码，并手动添加到路径，为了节省配置时间，本人采用了 vcpkg(Visual C++ Package Manager)一站式资源包管理器进行环境配置，配置过程如下：

1. 从 GitHub 上下载 vcpkg 源码

```
git clone https://github.com/Microsoft/vcpkg.git
```

2. 运行脚本文件将 vcpkg 配置到系统环境变量中

```
.\vcpkg\bootstrap-vcpkg.bat
```

3. 接着使用 vcpkg 进行资源库安装

```
vcpkg install glfw3:x64-windows glew:x64-windows glm:x64-windows soil2:x64-windows
```

4. 将 vcpkg 集成到 Visual Studio 开发环境中。执行此命令后，vcpkg 安装的库会自动配置到 Visual Studio 的项目中，无需手动设置包含路径、库路径或链接器设置。

```
vcpkg integrate install
```

至此，VS 中的 OpenGL 环境配置完成。

3 构建几何模型——环形

Paul Baker[2] 逐步描述了定义圆形切片，然后围绕圆圈旋转切片以形成环面的方法。图 1.1 描述了侧面和上面的两种视图。

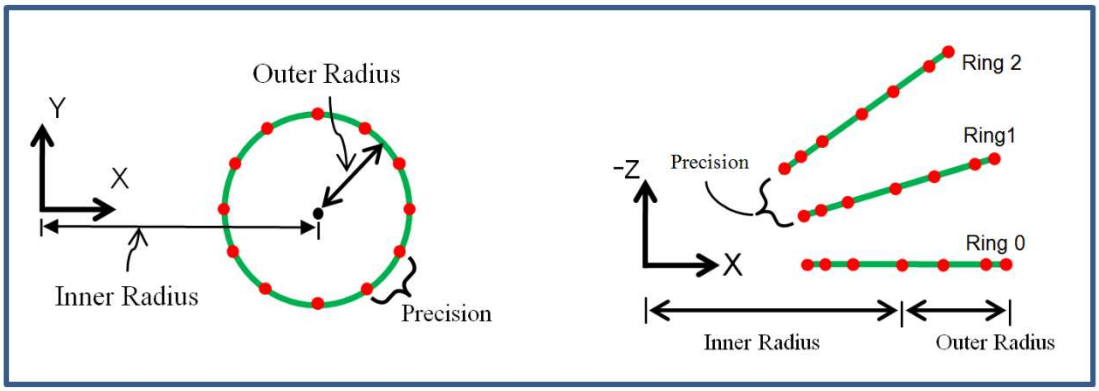


图 3.1 构建一个环面[1]

对于环面，在生成顶点坐标时，先将一个顶点定位到原点的右侧，然后在 XY 平面上的圆中让这个顶点围绕 Z 轴旋转，以形成“环”然后，将这个环“向外”移动“内半径”那么长的距离。在构建这些顶点时，为每个顶点计算纹理坐标和法向量。还会额外为每个顶点生成与环面表面相切的向量(称为切向量)。

围绕轴旋转最初的这个环，形成用来构成环面的其他环的顶点。通过围绕 Y 轴旋转最初的环的切向量和法向量来计算每个结果顶点的切向量和法向量。在顶点创建之后，逐个环地遍历所有顶点，并且对于每个顶点生成两个三角形。

为剩余的环选择纹理坐标的策略，将它们排列成使得纹理图像的 S 轴环绕环面的水平周边的一半，然后再对另一半重复。当我们绕 Y 轴旋转生成环时，我们指定一个从 1 开始并增加到指定精度的变量环（再次称为“pree”）。我们将 S 纹理坐标值设置为 ring 2.0prec，使 S 的取值范围介于 0.0 和 2.0 之间，然后每当纹理坐标大于 1.0 时减去 1.0。这种方法的动机是避免纹理图像在水平方向上过度“拉伸”。反之，如果我们确实希望纹理完全围绕环面拉伸，我们只须从纹理坐标计算中删除“*2.0”乘数即可。

本小节主要实现一个基于 Baker 策略的名为 Torus 的类，用于绘制圆环。“内”和“外”变量指的是图 3.2 中相应的内半径和外半径。prec 变量具有与球体类似的作用，对顶点数量和索引数量进行类似的计算。使用 Baker 描述中给出的策略，其中计算了两个切向量(Baker 称为 sTangent 和 tTangent,尽管通常称为“切向量(tangent)”和“副切向量(bitangent)”)，它们的叉乘积形成法向量。

Torus 类的实现逻辑如下程序所示。

- 属性:
  - int prec
  - float inner, outer
  - int numVertices, numIndices
  - vectors: vertices, texCoords, normals, sTangents, tTangents, indices
- 方法:
  - 构造函数:
    - Torus()
      - 调用 init()初始化

- Torus(float in, float out, int precIn)
    - 设置 inner, outer 和 prec
    - 调用 init()初始化
- float toRadians(float degrees):
    - 返回度数转换为弧度
- void init():
    - 计算顶点和索引数量
    - 初始化顶点、纹理坐标、法线、切线、副切线和索引的存储空间
    - 生成首个环的顶点和属性:
        - for i 从 0 到 prec + 1:
            - 计算旋转角度 amt
            - 使用旋转矩阵计算顶点初始位置 initPos
            - 存储顶点位置、纹理坐标、切线、副切线和法线
    - 生成其余环的顶点和属性:
        - for ring 从 1 到 prec + 1:
            - for i 从 0 到 prec + 1:
                - 计算旋转角度 amt
                - 使用旋转矩阵计算顶点位置
                - 存储顶点位置、纹理坐标、切线、副切线和法线
    - 计算三角形的索引:
        - for ring 从 0 到 prec:
            - for i 从 0 到 prec:
                - 计算并存储索引值
- 获取方法:
    - int getNumVertices(): 返回顶点数量
    - int getNumIndices(): 返回索引数量
    - vector<int> getIndices(): 返回索引
    - vector<glm::vec3> getVertices(): 返回顶点坐标
    - vector<glm::vec2> getTexCoords(): 返回纹理坐标
    - vector<glm::vec3> getNormals(): 返回法线
    - vector<glm::vec3> getStangents(): 返回切线
    - vector<glm::vec3> getTtangents(): 返回副切线

图 3.2 显示了实例化环面，并对表面上色的结果。

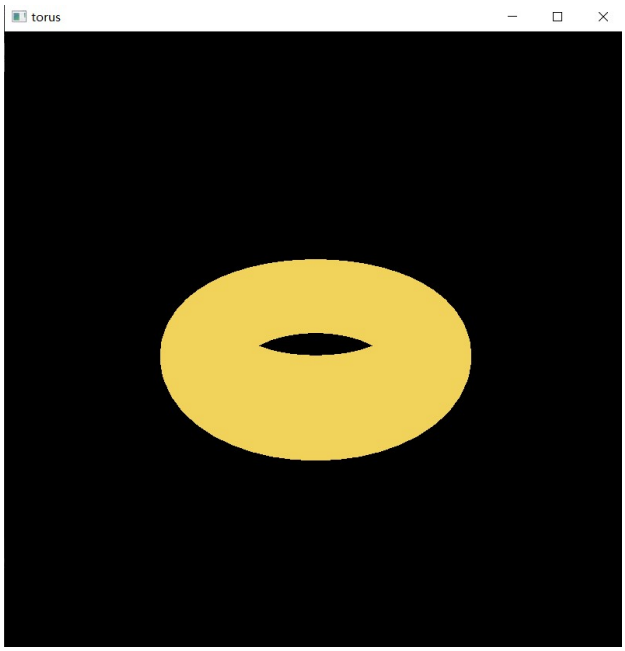


图 3.2 程序生成的环面

4 增加光照处理

如图 3.2 所生成的环面所示，生成的环面并不具备 3D 效果，对模型增加光照处理，可以物体在不同的角度和位置呈现出真实的阴影、反射和折射效果，来增强场景的深度感和立体感。另一方面，合适的光线和阴影处理，能够表现出物体的材质。

先前理论课已经讲述过基本的光照模型（ADS 模型）的基本知识，ADS 模型基于三种类型的反射：

- 环境光反射（Ambient）
- 漫反射（Diffuse）
- 镜面反射（Specular）

图 4.1 描述了 ADS 模型中的三种基本反射类型。

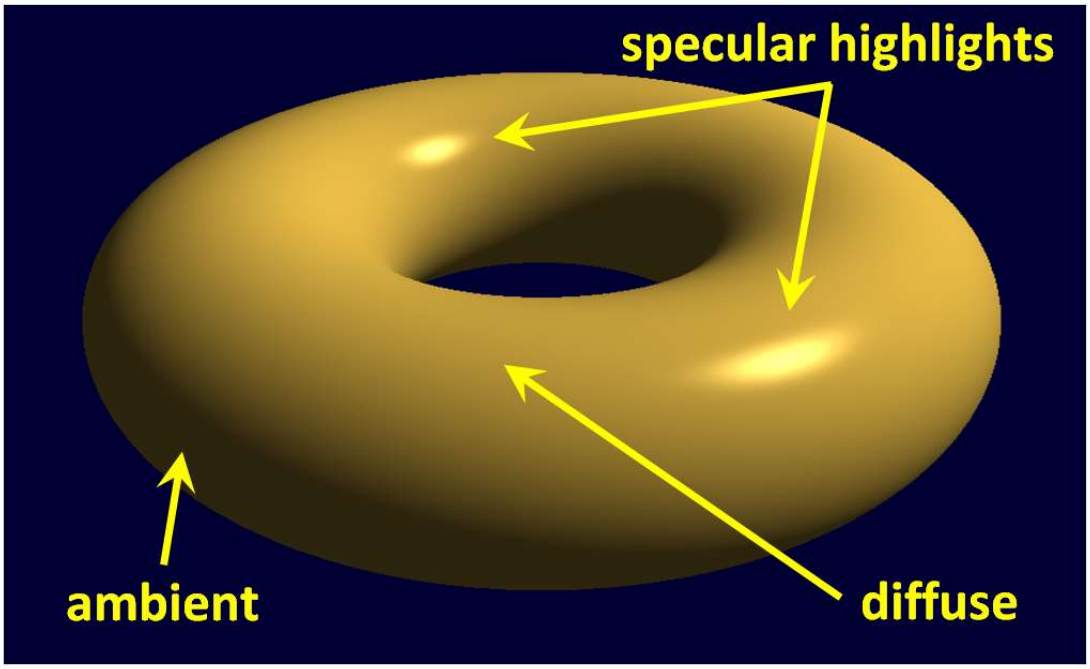


图 4.1 ADS 光照分量[1]

要在在环形中实现 ADS 光照，首先要对模型每个多边形的一个顶点进行光照计算，然后以每个多边形或每个三角形为基础，将计算结果的光照复制到相邻的像素中。为了使着色效果显得更加平滑，接下来将使用 Gouraud[3]着色（双线性光插值法），图 4.2 展示了在场景包含环面和单一位置光的情况下，在 OpenGL 中实现 Gouraud 着色器的策略。

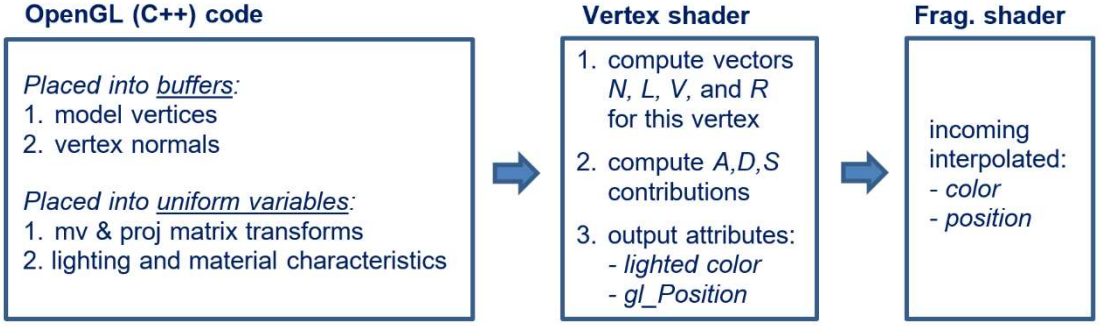


图 4.2 Gouraud 着色[1]

主程序实现逻辑如下。

```
初始化 OpenGL 和窗口
创建着色器程序对象并加载顶点和片段着色器
设置光照参数和材质参数
设置顶点数据
循环直到窗口关闭：
    清除颜色缓冲区和深度缓冲区
    使用着色器程序
    设置模型视图矩阵、投影矩阵和法线矩阵
    绑定顶点缓冲区和索引缓冲区
    绘制场景
    处理窗口事件
退出程序
```

核心函数 Display()

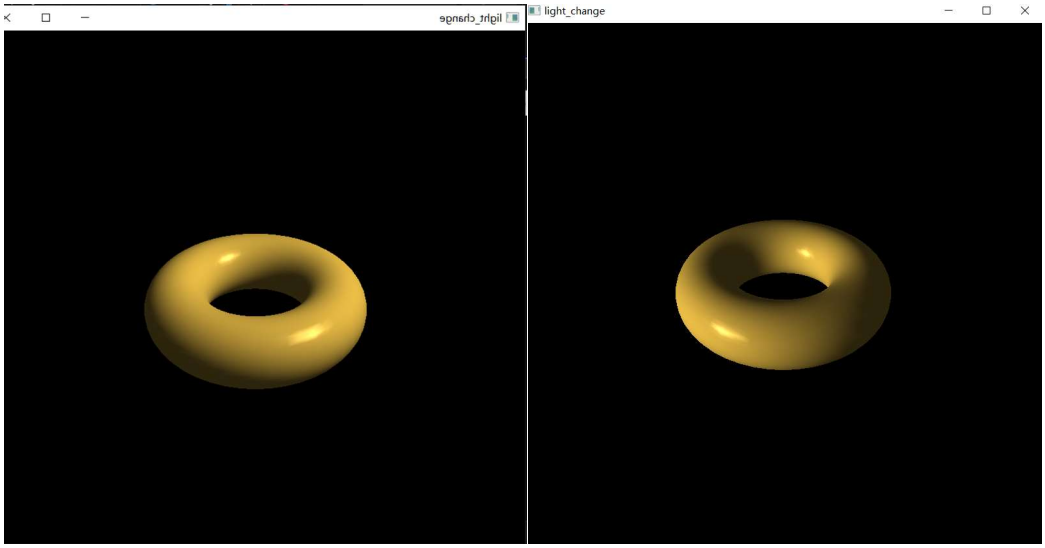
```
void display(GLFWwindow* window, double currentTime) {
    glClear(GL_DEPTH_BUFFER_BIT);
    glClear(GL_COLOR_BUFFER_BIT);
    glUseProgram(renderingProgram);
    mvLoc = glGetUniformLocation(renderingProgram, "mv_matrix");
    projLoc = glGetUniformLocation(renderingProgram, "proj_matrix");
    nLoc = glGetUniformLocation(renderingProgram, "norm_matrix");
    vMat = glm::translate(glm::mat4(1.0f), glm::vec3(-cameraX, -cameraY, -cameraZ));
    mMat = glm::translate(glm::mat4(1.0f), glm::vec3(torLocX, torLocY, torLocZ));
    mMat *= glm::rotate(mMat, toRadians(35.0f), glm::vec3(1.0f, 0.0f, 0.0f));
    currentLightPos = glm::vec3(initialLightLoc.x, initialLightLoc.y, initialLightLoc.z);
    amt = currentTime * 25.0f;
    rMat = glm::rotate(glm::mat4(1.0f), toRadians(amt), glm::vec3(0.0f, 0.0f, 1.0f));
    currentLightPos = glm::vec3(rMat * glm::vec4(currentLightPos, 1.0f));
    installLights(vMat);
    mvMat = vMat * mMat;
    invTrMat = glm::transpose(glm::inverse(mvMat));
    glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::value_ptr(mvMat));
    glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(pMat));
    glUniformMatrix4fv(nLoc, 1, GL_FALSE, glm::value_ptr(invTrMat));
    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, vbo[2]);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(1);
    glEnable(GL_CULL_FACE);
    glFrontFace(GL_CCW);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo[3]);
    glDrawElements(GL_TRIANGLES, numTorusIndices, GL_UNSIGNED_INT, 0);
}
```

首先，定义了环面、光照和材质特性。接着将环面顶点以及相关法向量读入缓冲区。diplay()函数将展示窗口中的渲染效果，display 函数同时将光照和材质信息传入顶点着色器。为了传入这些信息它调用 installLights 将光源在视觉空间中的位置，以及材质的 ADS 特性，读入相应的统一变量以供着色器使用。在此之前应提前定义了这些统一位置变量。其中一个重要的细节是变换矩阵 MV，用来将顶点位置移动到视觉空间，取 MV 的逆转置矩阵，在程序中，这个新增的矩阵叫作“invIrMat”，通过统一变量传入着色器。变量 lightPosV 包含光源在相机空间中的位置。我们每帧只需要计算一次，因此我们在 installLights()中[在 display()中调用]而非着色器中计算。着色器代码在下方所示。其中在顶点着色器最后进行了向量加法。

```
    #version 430
```

```
layout (location = 0) in vec3 vertPos;
layout (location = 1) in vec3 vertNormal;
out vec3 varyingNormal;
out vec3 varyingLightDir;
out vec3 varyingVertPos;
struct PositionalLight
{    vec4 ambient;
     vec4 diffuse;
     vec4 specular;
     vec3 position;
};
struct Material
{    vec4 ambient;
     vec4 diffuse;
     vec4 specular;
     float shininess;
};
uniform vec4 globalAmbient;
uniform PositionalLight light;
uniform Material material;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform mat4 norm_matrix;
void main(void)
{    varyingVertPos = (mv_matrix * vec4(vertPos,1.0)).xyz;
     varyingLightDir = light.position - varyingVertPos;
     varyingNormal = (norm_matrix * vec4(vertNormal,1.0)).xyz;
     gl_Position = proj_matrix * mv_matrix * vec4(vertPos,1.0);
}
```

图 4.3-图 4.6 展示了圆环在不同光源位置下的效果图，可以看到，光源位置不同和光源数量不同，圆环表面的阴影面积、高光点均不一样，圆环具备了更加强烈的立体感



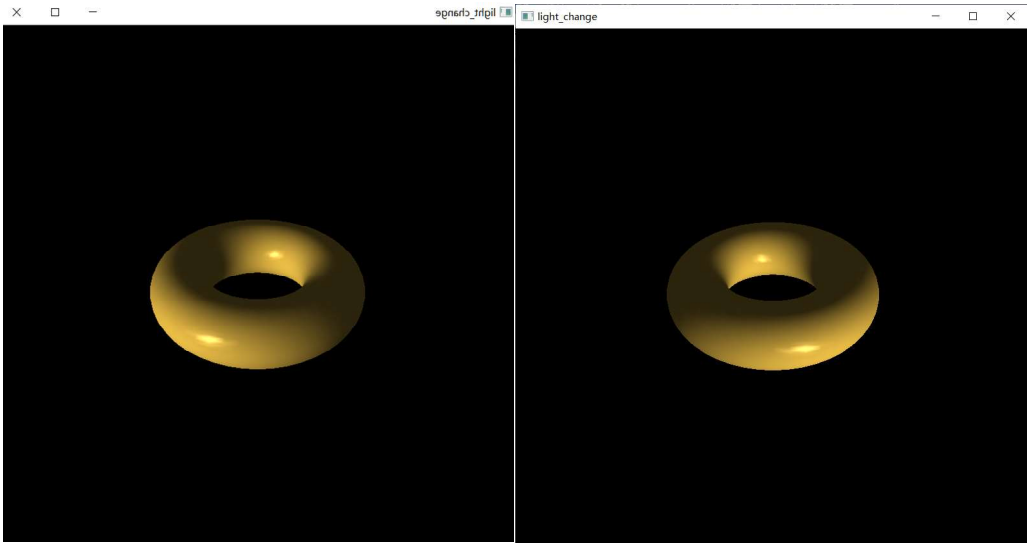


图 4.3-图 4.6 不同光源下圆环的不同光照效果

5 天空盒实现与模型贴图

5.1 天空盒实现

所谓天空盒，就是实例化一个立方体对象，然后将立方体的纹理设置为所需的环境，最后将立方体围绕相机放置，使得立方体能成功包围相机，来构成一种全景感。图 5.1 展示了天空盒的实现过程。

图 5.1 立体贴图包裹相机[1]

构建天空盒，本文使用 OpenGL 纹理立方体贴图法。使用 OpenGL 自带的 loadCubeMapO) 函数，读入 6 个单独的立方体面图像文件。图 5.2 展示了立方体 6 个面的纹理坐标。接着选择使用 SOIL2 库读取纹理图片，先找到需要读入的文件，然后调用 SOIL_lad_OGL_cubema() 函数，无须垂直翻转纹理，即可自动处理图像。

在片段着色器中使用名为 samplerCube() 的特殊类型的采样器访问纹理。在纹理立方体贴图中，从采样器返回的值是沿着方向向量(s, t, r)从原点“看到”的纹素。使用传入的插值项点位置作为纹理坐标。在顶点着色器中，将立方体顶点位置分配到输出纹理坐标属性中，以便在它们到达片段着色器时进行插值。核心函数代码如下文所示。

```
void display(GLFWwindow* window, double currentTime) {
    glClear(GL_DEPTH_BUFFER_BIT);
    glClear(GL_COLOR_BUFFER_BIT);
    processInput(window);
    Camera& currentCamera = cameras[currentCameraIndex];
    vMat = glm::lookAt(currentCamera.position, currentCamera.position + currentCamera.front,
currentCamera.up);
    pMat = glm::perspective(glm::radians(fov), aspect, 0.1f, 1000.0f);
    // 准备描绘天空盒
    glUseProgram(renderingProgramCubeMap);
    vLoc = glGetUniformLocation(renderingProgramCubeMap, "v_matrix");
    glUniformMatrix4fv(vLoc, 1, GL_FALSE, glm::value_ptr(vMat));
    projLoc = glGetUniformLocation(renderingProgramCubeMap, "proj_matrix");
```

```
    glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(pMat));
    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_CUBE_MAP, skyboxTexture);
    glEnable(GL_CULL_FACE);//禁用深度测试
    glFrontFace(GL_CCW);       // cube is CW, but we are viewing the inside
    glDisable(GL_DEPTH_TEST);
    glDrawArrays(GL_TRIANGLES, 0, 36);
    glEnable(GL_DEPTH_TEST);
    // draw scene (in this case it is just a torus)
    glUseProgram(renderingProgram);
    mvLoc = glGetUniformLocation(renderingProgram, "mv_matrix");
    projLoc = glGetUniformLocation(renderingProgram, "proj_matrix");
    nLoc = glGetUniformLocation(renderingProgram, "norm_matrix");
    rotAmt = currentTime * 0.5f;
    mMat = glm::translate(glm::mat4(1.0f), glm::vec3(torLocX, torLocY, torLocZ));
    mMat = glm::rotate(mMat, rotAmt, glm::vec3(1.0f, 0.0f, 0.0f));
    mvMat = vMat * mMat;
    invTrMat = glm::transpose(glm::inverse(mvMat));
    glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::value_ptr(mvMat));
    glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(pMat));
    glUniformMatrix4fv(nLoc, 1, GL_FALSE, glm::value_ptr(invTrMat));
    glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, vbo[2]);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(1);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_CUBE_MAP, skyboxTexture);
    glClear(GL_DEPTH_BUFFER_BIT);
    glEnable(GL_CULL_FACE);
    glFrontFace(GL_CCW);
    glDepthFunc(GL_LEQUAL);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo[3]);
    glDrawElements(GL_TRIANGLES, numTorusIndices, GL_UNSIGNED_INT, 0);
}
```

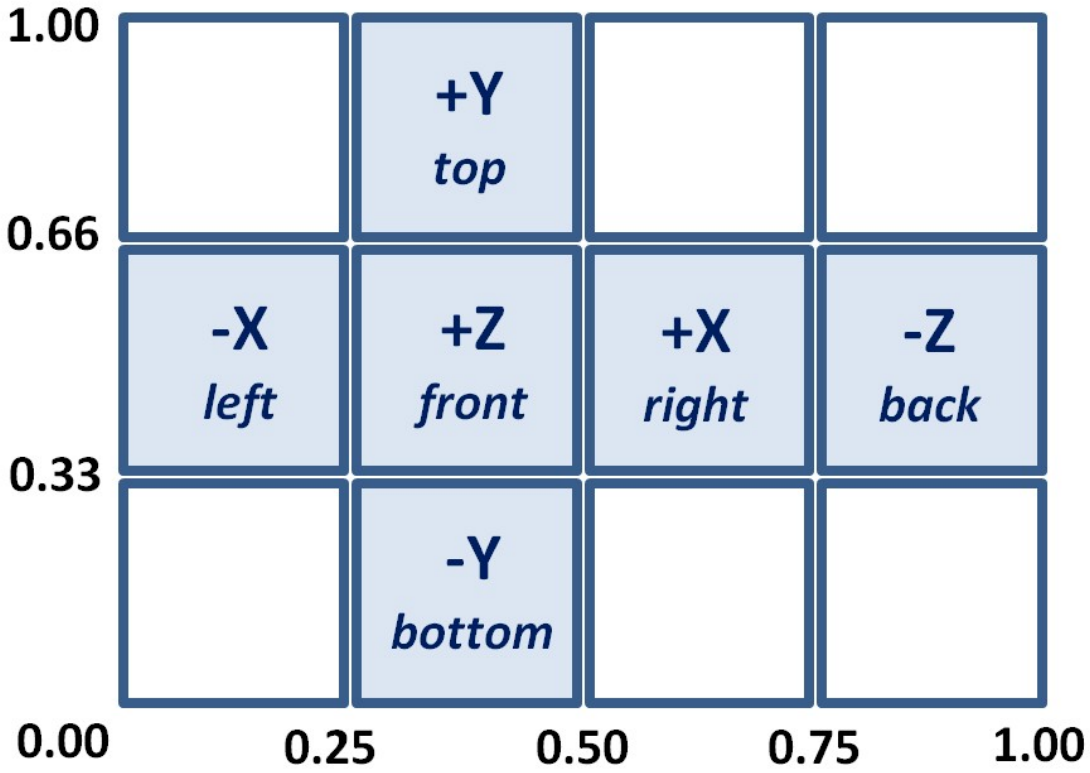


图 5.2 立方体贴图纹理坐标[1]

5.2 环境贴图实现

在第 4 小节中的进行光照设定时，我们了解到了物体表面的光泽度这个概念，对某一部分物体，如曲面镜，其表面有小范围高光的同时，还能反射出周围物体的镜像，这个时候 ASD 模型显然不适用了，为了使物体表面看起来更加真实，本文采用另外一种方法，使用视图向量和法向量组合计算反射向量，之后，该反射向量用来从立方体贴图中查找此元素，即环境贴图法。图 5.3 展示了环境贴图的总体流程。

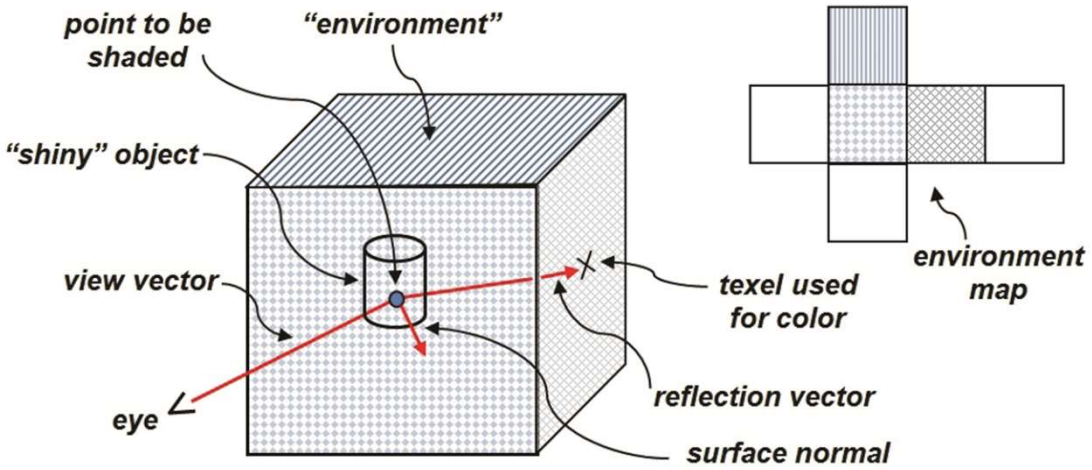


图 5.2 环境贴图总览[1]

在 5.1 的 Disply()函数中，只需要做很小的一部分修改，

- 创建用于变换法向量的矩阵(称为“norm matri”)并将其连接到关联的统一变量;

- 激活环面法向量缓冲区;
- 激活纹理立方体贴图为环面的纹理。

该场景需要两组着色器，一组用于里繁体贴图，另一组用于环面贴图，具体代码如下。

```
#version 430
layout (location = 0) in vec3 position;
out vec3 tc;
uniform mat4 v_matrix;
uniform mat4 proj_matrix;
layout (binding = 0) uniform samplerCube samp;
void main(void)
{
    tc = position;
    mat4 v3_matrix = mat4(mat3(v_matrix));
    gl_Position = proj_matrix * v3_matrix * vec4(position,1.0);
}

#version 430
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
out vec3 vNormal;
out vec3 vVertPos;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform mat4 norm_matrix;
layout (binding = 0) uniform samplerCube t;
void main(void)
{
    vVertPos = (mv_matrix * vec4(position,1.0)).xyz;
    vNormal = (norm_matrix * vec4(normal,1.0)).xyz;
    gl_Position = proj_matrix * mv_matrix * vec4(position,1.0);
}
```

图 5.3 展示了圆环在天空盒下的显示效果。可以看到该圆环的镜面质感，既有小范围的高光，又有对周围环境的倒映。

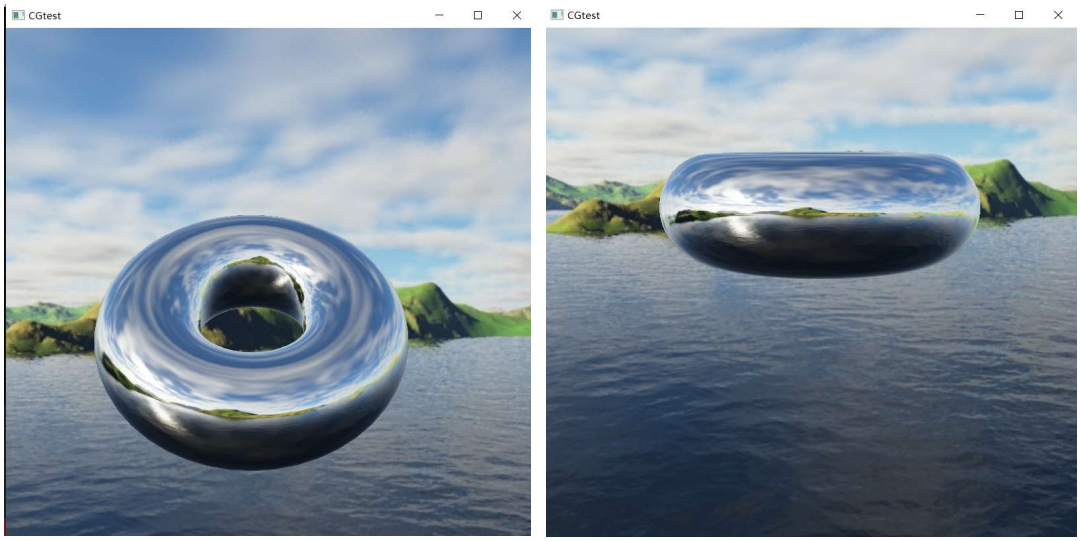


图 5.3 镜面圆环在天空盒下的显示效果图

6 交互设置

小节 5 实现了天空盒场景下的模型渲染与环境贴图，为了充分发挥天空盒场景的全景特性，必要的交互设置是不可少的，小节 6 通过增加鼠标回调函数，以及多摄像机设置来实现视角变换。

6.1 回调函数

1. 先定义鼠标操作所需要的全部变量。

```
float cameraSpeed = 0.05f;
float pitch = 0.0f;
float yaw = -90.0f;
float lastX = 400, lastY = 400;
bool firstMouse = true;
glm::vec3 cameraPos(0.0f, 0.0f, 5.0f);
glm::vec3 cameraFront(0.0f, 0.0f, -1.0f);
glm::vec3 cameraUp(0.0f, 1.0f, 0.0f);
float fov = 45.0f;
```

2. 设置鼠标回调函数，用于监听鼠标信号，函数代码如下。

```
void mouse_callback(GLFWwindow* window, double xpos, double ypos) {
    if (firstMouse) {
        lastX = xpos;
        lastY = ypos;
        firstMouse = false;
    }
    float xoffset = xpos - lastX;
    float yoffset = lastY - ypos; // reversed since y-coordinates go from bottom to top
```

```
        lastX = xpos;
        lastY = ypos;
        float sensitivity = 0.1f;
        xoffset *= sensitivity;
        yoffset *= sensitivity;
        yaw += xoffset;
        pitch += yoffset;
        if (pitch > 89.0f)
            pitch = 89.0f;
        if (pitch < -89.0f)
            pitch = -89.0f;
        glm::vec3 front;
        front.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
        front.y = sin(glm::radians(pitch));
        front.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
        cameraFront = glm::normalize(front);
}
```

3. 在 Display() 中添加更新的相机前向向量和视野（FOV）。

```
glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
glBindBuffer(GL_ARRAY_BUFFER, vbo[2]);
```

4. 在 main() 函数中调用鼠标回调函数即可。

6.2 多机位摄像

1. 增加一个数组来存储多个摄像机位置以及一个变量来追踪当前摄像机位置索引。

```
// Camera settings
struct Camera {
    glm::vec3 position;
    glm::vec3 front;
    glm::vec3 up;
    float yaw;
    float pitch;
};
std::vector<Camera> cameras;
```

2. 在 init() 初始化函数中增加多个摄像机的位置，本文采用 4 个机位进行测试。

```
    // Initialize multiple camera positions
    cameras.push_back({glm::vec3(0.0f, 0.0f, 5.0f), glm::vec3(0.0f, 0.0f, -1.0f), glm::vec3(0.0f, 1.0f, 0.0f), -90.0f, 0.0f});
    cameras.push_back({glm::vec3(5.0f, 5.0f, 5.0f), glm::vec3(-1.0f, -1.0f, -1.0f), glm::vec3(0.0f, 1.0f, 0.0f), -135.0f, -35.0f});
    cameras.push_back({glm::vec3(-5.0f, 5.0f, 5.0f), glm::vec3(1.0f, -1.0f, -1.0f), glm::vec3(0.0f, 1.0f, 0.0f), 135.0f, -35.0f});
    cameras.push_back({glm::vec3(0.0f, 5.0f, 0.0f), glm::vec3(0.0f, -1.0f, -1.0f), glm::vec3(0.0f,
```

```
1.0f, 0.0f), -90.0f, -45.0f});
```

3. 在 display() 函数中使用当前摄像机的位置和方向。
4. 添加右键切换摄像机功能。

```
void mouse_button_callback(GLFWwindow* window, int button, int action, int mods) {
    if (button == GLFW_MOUSE_BUTTON_RIGHT && action == GLFW_PRESS) {
        currentCameraIndex = (currentCameraIndex + 1) % cameras.size();
    }
}
```

5. 在 main 函数中启用添加的功能。

图 6.1-图 6.4 展示了不同视角的摄像机的视角，同时，通过鼠标移动，能看到 360° 的全景视图（具体的演示视频可以查看代码压缩包）。

图 6.1-图 6.4 不同视角下的天空盒场景

7 实验总结

本文通过对圆环这一基本几何体进行建模，完成了包括绘制、添加光照、设置天空盒、环境贴图、交互设置等建模基本流程，笔者在体验到图形学带来的趣味的同时，对理论课学到的知识也有了更深的理解。本文大部分内容都参考书籍“Computer Graphics Programming in OpenGL with C++”。文中重点介绍实现的原理思路，关于代码只介绍实现逻辑和部分核心代码，完整代码可以查看压缩包，具体的演示效果也可以查看演示视频.mp4。

参考文献

[1] V. Scott Gordon, John Clevenger. Computer Graphics Programming in OpenGL with C++[M]. 1. 人民邮电出版社, 2022.02.

[2] P. Baker, *Paul's Projects*, 2007, accessed October 2018.

[3] H. Gouraud, "Continuous Shading of Cured Surfaces," *IEEE Transactions on Computers* C-20, no. 6(June 1971).